Further Remarks on DNA Overlap Assembly[☆]

Srujan Kumar Enaganti[a], Oscar H. Ibarra[b], Lila Kari[a,c,*], Steffen Kopecki[a]

[a]*Department of Computer Science, University of Western Ontario, London, ON, N6A 5B7, Canada*
[b]*Department of Computer Science, University of California Santa Barbara, Santa Barbara, CA, 93106, USA*
[c]*School of Computer Science, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*

Abstract

The operation of *overlap assembly* was defined by Csuhaj-Varju, Petre, and Vaszil as a formal model of the linear self-assembly of DNA strands: The overlap assembly of two strings, xy and yz, which share an "overlap" y, results in the string xyz. This paper continues the exploration of the properties of the overlap assembly operation by investigating closure of various language classes under iterated overlap assembly, and the decidability of the completeness of a language. It also investigates the problem of deciding whether a given string is terminal with respect to a language, and the problem of deciding if a given language can be generated by an overlap assembly operation of two given others.

Keywords: DNA computing, bio-operations, DNA self-assembly, overlap assembly

1. Introduction

The word and language operation *overlap assembly* was first introduced by Csuhaj-Varju, Petre, and Vaszil - under the name (self-)assembly - in [1], and later studied in [2], as a formal model of the linear self-assembly of DNA strands. Formally, the overlap assembly is a binary operation which, when

[☆]This research was supported by Natural Science and Engineering Council of Canada (NSERC) Discovery Grant R2824A01 and a University of Western Ontario Grant to L.K., and US NSF Grant CCF-1117708 to O.H.I.

[*]Corresponding author
Email address: lila.kari@uwo.ca (Lila Kari)

applied to two input strings xy and yz (where y is their nonempty *overlap*), produces the output xyz.

The study of overlap assembly as a formal language operation is part of ongoing efforts to provide a formal framework and rigorous treatment of DNA-based information and DNA-based computation. More specifically, this study can be placed in the context of studies of DNA bio-operations enabled by the action of the DNA polymerase enzyme, such as hairpin completion and its inverse operation, hairpin reduction [3, 4, 5, 6], overlapping concatenation [7], and directed extension [8].

The activity of DNA Polymerase presupposes the existence of a DNA single strand, called *template*, and of a second short DNA strand, called *primer*, that is Watson-Crick complementary to the template and binds to it. Given a supply of individual nucleotides, DNA polymerase then extends the primer, at one of its ends only, by adding individual nucleotides complementary to the template nucleotides, one by one, until the end of the template is reached. In the wet lab, the iteration of this process is used to obtain an exponential replication of DNA strands, in a protocol called *Polymerase Chain Reaction (PCR)*. Experimentally, (parallel) overlap assembly of DNA strands under the action of the DNA Polymerase enzyme was used for gene shuffling in, e.g., [9]. In the context of experimental DNA computing, overlap assembly was used in, e.g., [10, 11, 12, 13] for the formation of combinatorial DNA or RNA libraries. Overlap assembly can also be viewed as modelling a special case of an experimental procedure called cross-pairing PCR, introduced in [14] and studied in, e.g., [15, 16, 17, 18].

This paper is a continuation of the theoretical analysis of overlap assembly as a formal language operation, that was started in [1] and [2]. In [1], the authors proposed a formal language operation called "self-assembly", inspired by the linear self-assembly of DNA single strands via Watson-Crick complementarity. The authors obtained closure properties of various language classes under iterated overlap assembly, and studied the question of whether or not a given language can be generated through assembly and, if so, what is the minimal generator. In [2], the aforementioned formal operation of self-assembly was renamed overlap assembly, to avoid confusion with other usages of the syntagm self-assembly, such as in the context of DNA computing by two-dimensional self-assembly of DNA rectangular tiles, [19, 20, 21]. The paper [2] explored closure properties of basic language families under overlap assembly, decision problems, as well as the potential use of iterated overlap assembly to generate combinatorial DNA libraries.

In this paper, following Section 2 comprising definitions, notations and basic properties, in Section 3 we correlate the overlap assembly operation with the superposition operation introduced in [22] and determine additional closure properties of some language classes under iterated overlap assembly. A string w is terminal with respect to a language L if $w \in L$ and the result of the overlap assembly between w and L equals $\{w\}$; the terminal set $T(L)$ contains all words that are terminal with respect to L. In Section 4 we investigate the closure properties of terminal sets of complete languages (languages closed under overlap assembly). In Section 5 we study three decision problems: deciding the completeness of an arbitrary language, deciding whether a string is terminal with respect to a language, and deciding whether a language is generated by an overlap assembly operation between two given languages. Section 6 contains concluding remarks.

2. Basic definitions and notations

An alphabet Σ is a finite nonempty set of symbols. Σ^* denotes the set of all words over Σ, including the empty word λ. Σ^+ is the set of all nonempty words over Σ. For words $w, x, y, z \in \Sigma^*$ such that $w = xyz$ we call the subwords x, y, and z *prefix*, *infix*, and *suffix* of w, respectively. The sets $\text{pref}(w)$, $\text{inf}(w)$, and $\text{suff}(w)$ contain, respectively, all prefixes, infixes, and suffixes of w. A prefix (resp., infix or suffix) x of w is *proper* if $x \neq w$. We employ the following notation: $\text{Pref}(w) = \text{pref}(w) \setminus \{w\}$, $\text{Inf}(w) = \text{inf}(w) \setminus \{w\}$, and $\text{Suff}(w) = \text{suff}(w) \setminus \{w\}$. This notation is naturally extended to languages; for example, $\text{Suff}(L) = \bigcup_{w \in L} \text{Suff}(w)$. The complement of a language $L \subseteq \Sigma^*$ is $L^c = \Sigma^* \setminus L$.

Let $\mathbb{N}$ be the set of non-negative integers and k be a positive integer. A subset Q of $\mathbb{N}^k$ is a *linear set* if there exist vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_n \in \mathbb{N}^k$ such that $Q = \{\vec{v}_0 + i_1\vec{v}_1 + \cdots + i_n\vec{v}_n \mid i_1, \ldots, i_n \in \mathbb{N}\}$. A finite union of linear sets is called a *semilinear set*.

Let $\Sigma = \{a_1, \ldots, a_k\}$. The *Parikh map* of a language $L \subseteq \Sigma^*$, denoted $\Psi(L)$, is defined as

$$\Psi(L) = \{(|w|_{a_1}, \ldots, |w|_{a_k}) \mid w \in L\},$$

where $|w|_{a_i}$ is the number of a_i's in w.

2.1. The overlap assembly

An *involution* is a function $\theta : \Sigma^* \to \Sigma^*$ with the property that θ^2 is the identity. θ is called an *antimorphism* if $\theta(uv) = \theta(v)\theta(u)$. Traditionally, the Watson-Crick complementarity of DNA strands has been modeled as an antimorphic involution over the DNA alphabet $\Delta = \{A, C, G, T\}$, [23, 24].

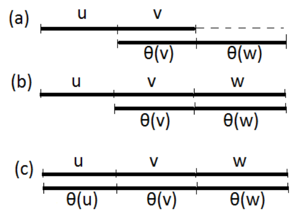

Figure 1: (a) The two input DNA single-strands, uv and $\theta(w)\theta(v)$ bind to each other through their complementary segments v and $\theta(v)$, forming a partially double-stranded DNA complex. (b) DNA polymerase extends the 3' end of the strand uv. (c) DNA polymerase extends the 3' end of the other strand. The resulting DNA double strand is considered to be the output of the *overlap assembly* of the two input single strands.

Using the convention that a word x over this alphabet represents the DNA single strand x in the 5' to 3' direction, the overlap assembly of a strand uv with a strand $\theta(w)\theta(v)$ first forms a partially double-stranded DNA molecule with v in uv and $\theta(v)$ in $\theta(w)\theta(v)$ attaching to each other, see Figure 1(a). The DNA polymerase enzyme will extend the 3' end of uv with the strand w, see Figure 1(b). Similarly, the 3' end of $\theta(w)\theta(v)$ will extended, resulting in a full double strand whose upper strand is uvw, see Figure 1(c). Formally, the overlap assembly between uv and $\theta(w)\theta(v)$ is uvw. Assuming that all involved DNA strands are initially double-stranded, that is, whenever the strand x is available, its Watson-Crick complement $\theta(x)$ is also available, one can further simplify this model and, given two words x, y over an alphabet Σ, define the *overlap assembly of x with y*, [1], as:

$$x \overline{\odot}\, y = \{z \in \Sigma^+ \mid \exists u, w \in \Sigma^*, \exists v \in \Sigma^+ : x = uv, y = vw, z = uvw\}.$$

The definition of overlap assembly can be extended to languages in the natural way.

The *iterated overlap assembly* $\mu_*(L)$ of a language L is defined as:

$$\mu_0(L) = L, \qquad \mu_{i+1}(L) = \mu_i(L) \overline{\odot}\, \mu_i(L), \qquad \mu_*(L) = \bigcup_{i \geq 0} \mu_i(L).$$

Since $w \in w \overline{\odot}\, w$ for any nonempty word w, it easily follows that $\mu_i(L) \subseteq \mu_{i+1}(L)$ for $L \in \Sigma^+$.

A string $w \in L$ is said to be *terminal* with respect to the language L if $w \,\overline{\odot}\, L = L \,\overline{\odot}\, w = \{w\}$. A set of strings $T(L) \subseteq L$ is said to be the *(maximal) terminal set* of L if every $w \in T(L)$ is terminal with respect to L and for all $w \in L \backslash T(L)$, w is not terminal with respect to L, that is,

$$T(L) = \{w \in L \mid w \,\overline{\odot}\, L = L \,\overline{\odot}\, w = \{w\}\}.$$

A *complete* language is a language closed under overlap assembly, that is, $L \,\overline{\odot}\, L = L$. For every language L, the language $\mu_*(L)$ is complete and $\mu_*(L) = L$ if and only if L is complete. The investigation of the terminal set $T(L)$ makes most sense if L is complete, but it is well-defined for all languages L.

2.2. Basic properties of the overlap assembly

In this section we present a few basic properties of the (iterated) overlap assembly and complete languages. The operation of overlap assembly is not associative, as seen in the next example.

The following example shows that overlap assembly is not associative

$$(aba \,\overline{\odot}\, a) \,\overline{\odot}\, bac = \{abac\}, \qquad\qquad aba \,\overline{\odot}\, (a \,\overline{\odot}\, bac) = \emptyset.$$

The words in the example are chosen such that a and bac cannot form an overlap; note that, however, $aba \,\overline{\odot}\, bac = (aba \,\overline{\odot}\, a) \,\overline{\odot}\, bac$ in this example.

However, overlap assembly does satisfy a property related to associativity, as seen in the following result.

Lemma 2.1. *For languages L_x, L_y and L_z, we have $((L_x \,\overline{\odot}\, L_y) \,\overline{\odot}\, L_z) \cup (L_x \,\overline{\odot}\, L_z) = (L_x \,\overline{\odot}\, (L_y \,\overline{\odot}\, L_z)) \cup (L_x \,\overline{\odot}\, L_z)$.*

Proof. Consider a word $w \in ((L_x \,\overline{\odot}\, L_y) \,\overline{\odot}\, L_z) \cup (L_x \,\overline{\odot}\, L_z)$. If $w \in (L_x \,\overline{\odot}\, L_z)$, then w clearly belongs to $(L_x \,\overline{\odot}\, (L_y \,\overline{\odot}\, L_z)) \cup (L_x \,\overline{\odot}\, L_z)$. Otherwise, there exist $x \in L_x$, $y \in L_y$ and $z \in L_z$ such that $w = (x \,\overline{\odot}\, y) \,\overline{\odot}\, z$. Because $w \notin x \,\overline{\odot}\, z$, we can factorize $y = uy'v$ such that $w = xy'z$, u is a nonempty suffix of x, and v is a nonempty prefix of z. We conclude $uy'z \in (y \,\overline{\odot}\, z)$ and $w = xy'v \in x \,\overline{\odot}\, (y \,\overline{\odot}\, z)$. The converse inclusion follows by similar arguments. □

The next lemma considers a word $w \in \mu_*(L)$ and its infixes that belong to L. It is not difficult to see that the entire word w is "covered" by overlapping words from L. We formalize this observation by saying that every two consecutive letters in the word w are covered by an infix v of w that belongs to L.

Lemma 2.2. *Let $L \in \Sigma^*$ be any language and $w = a_1a_2 \cdots a_n \in \mu_*(L)$ for letters $a_1, a_2, \ldots, a_n \in \Sigma$. For each integer k with $1 \leq k < n$ there exist integers j, ℓ such that $1 \leq j \leq k < \ell \leq n$ and $a_ja_{j+1} \cdots a_\ell \in L$.*

Proof. The statement is trivially true if $|w| = n < 2$ or $w \in \mu_0(L) = L$. By induction, assume that the statement holds for all words in $\mu_i(L)$ and consider $w \in \mu_{i+1}(L) \setminus \mu_i(L)$. We have $w \in x \overline{\odot} y$ for some $x, y \in \mu_i(L)$. If $k < |x|$, we find j, ℓ such that $1 \leq j \leq k < \ell \leq |x|$ and $a_ja_{j+1} \cdots a_\ell \in L$ because x is a proper prefix of w. Here, we know that we can always find j and ℓ because of the inductive hypothesis that the statement is valid for all $x \in \mu_i(L)$. If $k \geq |x| \geq n - |y| + 1$, we find j, ℓ such that $n - |y| + 1 \leq j \leq k < \ell \leq n$ and $a_ja_{j+1} \cdots a_\ell \in L$ because y is a proper suffix of w. Again, we are using the inductive hypothesis that the statement is valid for all $y \in \mu_i(L)$. $\square$

Lemma 2.2 leads to the following statement, that allows us to identify non-terminal words in a language $\mu_*(L)$.

Lemma 2.3. *For a language L and a word $w \in \mu_*(L) \setminus T(\mu_*(L))$, there exist $z \in \mu_*(L) \setminus \{w\}$ and $x \in L$ such that $z \in w \overline{\odot} x$ or $z \in x \overline{\odot} w$.*

Proof. A word $w \in \mu_*(L)$ is not terminal with respect to $\mu_*(L)$ if there exists $y \in \mu_*(L)$ and $v \neq w$ such that $v \in w \overline{\odot} y$ or $v \in y \overline{\odot} w$. Due to symmetry, we only consider the case when $v \in w \overline{\odot} y$. Let $y = y_1y_2$ such that $v = wy_2$ and y_1 is a suffix of w. We will use Lemma 2.2 with the following interpretation: since $y = y_1y_2 \in \mu_*(L)$, there are $x_1, x_2 \in \Sigma^+$ with $x = x_1x_2 \in L$ such that x_1 is a suffix of y_1 and x_2 is a prefix of y_2. Let $z = wx_2$ and observe that $z \in w \overline{\odot} x$ and $z \in \mu_*(L) \setminus \{w\}$. $\square$

Finally, let us state a simple observation on complete languages.

Lemma 2.4. *For any $L \subseteq \Sigma^*$ and $\# \notin \Sigma$, the languages $\#L$ and $L\#$ are complete.*

Proof. Since every word in $\#L$ contains exactly one letter $\#$, this letter has to match in an overlap assembly. Therefore, $\#w \in \#x \overline{\odot} \#y$ for $x, y, w \in \Sigma^+$ if and only if x is a prefix of y and $w = y$. $\square$

2.3. Automata models, augmented with counters

We will use the following notations for language acceptors:

- DFA (NFA) = deterministic (nondeterministic) finite automaton. DFAs and NFAs are equivalent, and they accept exactly the regular languages.
- DPDA (NPDA) = deterministic (nondeterministic) pushdown automaton, i.e., a DFA (NFA) augmented with a pushdown stack. NPDAs accept exactly the context-free languages (CFLs).
- DCA (NCA) is a DPDA (NPDA) which uses only one stack symbol in addition to the bottom stack symbol which is never erased. DCAS (NCAs) are strictly weaker than DPDAs (NPDAs).
- DLBA (NLBA) = deterministic (nondeterministic) linear-bounded automaton, i.e., a two-way deterministic (nondeterministic) finite automaton that can read and re-write the tape. The machines have left and right input end markers. NLBAs accept exactly the context-sensitive languages (CSLs). It is a long-standing open problem whether or not DLBAs are strictly weaker than NLBAs.
- DTM (NTM) = deterministic (nondeterministic) Turing machines. DTMs and NTMs are equivalent, and they accepts exactly the recursively enumerable languages. Halting DTMs and NTMs are equivalent, and they accept exactly the recursive languages.

We refer the reader to [25] for the formal definitions of these devices.

A *counter* is an integer variable that can be incremented by 1, decremented by 1, left unchanged, and tested for zero. It starts at zero and cannot store negative values. Thus, a counter is a pushdown stack on a unary alphabet, in addition to the bottom of the stack symbol which is never altered. Note that a DCA (NCA) is equivalent to a DFA (NFA) which is augmented with a counter.

An automaton (DFA, NFA, DPDA, NPDA, DCA, NCA, etc.) can be augmented with a finite number of counters, where the "move" of the machine also now depends on the status (zero or non-zero) of the counters, and the move can update the counters. It is well known that a DFA augmented with two counters is equivalent to a DTM [26].

In this paper, we will restrict the augmented counter(s) to be reversal-bounded in the sense that each counter can only reverse (i.e., change mode from non-decreasing to non-increasing and vice-versa) at most r times for some given r. In particular, when $r = 1$, the counter reverses only once, i.e., once it decrements, it can no longer increment. Note that a counter that makes r reversals can be simulated by $\lceil \frac{r+1}{2} \rceil$ 1-reversal counters [27]. Closure and decidable properties of various machines augmented with reversal-bounded counters have been studied in the literature (see, e.g., [27, 28]). We will use the notation DFCM, NFCM, DPCM, NPCM, DNCM, NFCM, etc., to denote a DFA, NFA, DPDA, NPDA, DCA, NCA, etc., augmented with reversal-bounded counters.

Automata with reversal-bounded counters can "count", as seen in the following example.

Example 2.1. $L = \{xx^r \mid x \in \{a,b\}^+, |x|_a = |x|_b\}$ can be accepted by an NPCM M with two 1-reversal counters. Briefly, M operates as follows: It scans the input and uses the pushdown stack to check that the input is a palindrome (this requires M to "guess" the middle of the string) while using two counters C_1 and C_2 to store the numbers of a's and b's it encounters. Then, at the end of the input, on λ-transitions (i.e., without reading any input symbol), M decrements C_1 and C_2 simultaneously and verifies that they become zero at the same time. Note that the counters are 1-reversal.

A nondetermnistic stack automaton (NSA) is a generalization of an NPDA in that during the computation, the machine can enter the stack in a read-only mode. It can only push and pop by returning to the top of the stack [29]. The deterministic version is denoted by DSA.

A nondeterministic stack-counter automaton (NSCA) is a special case of an NSA in that the the stack alphabet is unary, except for the bottom of the stack symbol which is never altered and only used for this purpose [30]. So, again, the machine can enter the unary storage in a read-only mode, i.e., it can only go up and down the unary storage and not alter the stack. It can only push or pop by returning to the top of the stack. So the (unary) stack height can vary during the computation. The deterministic version is denoted by DSCA.

An NSCA (DSCA) is strictly more powerful than an NCA (DCA) as the following example shows.

Example 2.2.

1. $L_1 = \{a^{nk} \mid n, k \geq 1\}$ can be accepted by an NSCA M_1 which on λ-moves first writes Ba^n on the stack, where B is the bottom of the stack symbol and $n \geq 1$ is nondeterministically chosen. Then M_1 goes up

and down the stack (on a read-only mode) and checks that the length of the input is nk for some k. Note that after M_1 has written Ba^n, it will no longer pop or write on the stack.

2. $L_2 = \{a^1\#a^2\# \cdots \#a^k \mid k \geq 1\}$ can be accepted by a DSCA M_2 which, on a λ-move first writes B (the bottom of the stack symbol) on the stack. Then for each $1 \leq i \leq k$, M_1 pushes a on the stack (i.e., increases the length by 1) and then go down the stack and checks that the next input segment a^i is exactly the the numbers of a's on the stack. Note that M_2 does not do any popping.

The languages L_1 and L_2 above cannot be accepted by NCAs, since these languages have non-semilinear Parikh maps but languages accepted by NCAs (in fact, by NPCMs) have semilinear Parikh maps [27].

NSPACE($S(n)$) and NTIME($T(n)$) denote the classes of languages accepted by nondeterministic Turing machines in $S(n)$ space and $T(n)$ time, respectively. DSPACE($S(n)$) and DTIME($T(n)$) are the the corresponding deterministic classes. PTIME denotes the class of languages accepted by deterministic Turing machines in polynomial time.

3. The related superposition operation

The superposition operation is a binary operation proposed by Bottoni, Labella, Manca, and Mitrana in [22] to model the action of the DNA Polymerase enzyme. The result of the superposition operation between words $x, y \in \Sigma^+$, denoted by $x \diamond y$, consists of the set of all words $z \in \Sigma^+$ obtained by any of the four following cases ($^-$ denotes the *morphic complement*, that is, $^-$ is a morphism such that $\overline{\overline{u}} = u$ for all words u):

1. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = uw, y = \overline{w}v$, then $z = uw\overline{v} \in x \diamond_1 y$.

2. If there exist $u, v \in \Sigma^*$ such that $x = u\overline{y}v$, then $z = u\overline{y}v \in x \diamond_2 y$.

3. If there exist $u, v \in \Sigma^*, w \in \Sigma^+$ such that $x = wv, y = u\overline{w}$, then $z = \overline{u}wv \in x \diamond_3 y$.

4. If there exist $u, v \in \Sigma^*$ such that $y = u\overline{x}v$, then $z = \overline{u}x\overline{v} \in x \diamond_4 y$.

The superposition operation can be naturally extended to languages. For more on the superposition operation between two words (languages), the reader is referred to [22, 31].

The superposition operation and the overlap assembly are closely related. In particular, when we replace the complement $^-$ by the identity, then case 1 above is identical to overlap assembly, i.e., $x \,\overline{\odot}\, y = x \diamond_1 y$; case 3 above is symmetrical to the overlap assembly, i.e., $x \,\overline{\odot}\, y = y \diamond_3 x$; furthermore, cases 2 and 4 above give $x \diamond_2 y = y \diamond_4 x = x$ if y is an infix of x. However, in the general case of two languages or when we consider a "real" complement function, the overlap assembly $L_x \,\overline{\odot}\, L_y$ does not give the same result as the superposition $L_x \diamond L_y$.

If $^-$ is the identity, then the overlap assembly of a language L with itself gives $\mu_1(L) = L \,\overline{\odot}\, L = L \diamond L$ and, moreover, the iterated overlap assembly coincides with the analogously defined[1] iterated superposition $\mu_*(L) = \diamond^*(L)$. In other words, the iterated overlap assembly is a special case of the iterated superposition. Therefore, the (positive) closure results for the iterated superposition, obtained in [22, 31], also hold for the iterated overlap assembly. Indeed, the same results were independently obtained in [1], in its study of closure properties of iterated overlap assembly:

Proposition 1 ([1]). *The language classes of regular, context-sensitive, recursive and recursively enumerable languages are closed under iterated overlap assembly.*

The result that the family of context-free languages is not closed under iterated overlap assembly (resp., iterated superposition) was proven in [1] (resp., [22]). The following statement strengthens that result.

Theorem 3.1. *There is a language L accepted by a 1-reversal DCA (i.e., a DFA with one counter that makes only 1 reversal) whose iterated overlap assembly, $\mu^*(L)$, cannot be accepted by any NPCM (i.e., an NPDA with reversal-bounded counters).*

Proof. Let $L = \{\$a^i\#\$a^{i+1}\# \mid i \geq 1\}$. Clearly, L can be accepted by a 1-reversal DCA. Suppose $\mu_*(L)$ could be accepted by an NPCM. Then $\mu_*(L) \cap$

[1]When $^-$ is not the identity, the iterated superposition has to be defined slightly different than the iterated overlap assembly, because $L \subseteq L \diamond L$ does not necessarily hold. In [22, 31] two iterated versions of the superposition are defined which turn out to yield the same language.

$\$a\#(\$a^+\#)^+ = \{\$a^1\#\$a^2\#\cdots\$a^k\#\$a^{k+1}\# \mid k \geq 1\}$ could also be accepted by an NPCM. This is not possible since the Parikh map of this language is not semilinear, but it is known that the Parikh map of any NPCM language is semilinear [27]. $\square$

In contrast to the previous result, for any k and NFCM language L, $\mu_k(L)$ can be accepted by an NFCM. This follows from the fact that $L \overline{\odot} L$ is an NFCM language if L is an NFCM language [2].

In [22], the notion of a *maximal (adult) language* is defined, which is analogous to the terminal set of a language. The authors consider *maximal (adult) words* with respect to the iterated superposition of some language L: A word x is a maximal word with respect to $\diamond^*(L)$ if $x \in \diamond^*(L)$ and $x \diamond (\diamond^*(L)) \subseteq \{x\}$. Also, *max* $\diamond^*$ (L) is defined as the set of all maximal words with respect to $\diamond^*(L)$. We immediately obtain that $T(\mu_*(L))$ is the special case of *max* $\diamond^*$ (L) where the complement $^-$ is replaced by the identity. From the results in [22, 31] we obtain the following result for terminal sets of complete, regular languages:

Proposition 2. *The terminal set $T(L)$ of a complete, regular language L is regular. In particular, the terminal set $T(\mu_*(L))$ is regular if L is regular.*

It is known from [22] that there exists a context-sensitive language L such that the maximal language *max* $\diamond^*$ (L) is undecidable. This result can be strengthened as follows.

Theorem 3.2. *There exists a (complete) language $L \in \mathrm{DSPACE}(\log n)$, such that $T(L)$ and $T(\mu_*(L))$ are undecidable.*

Proof. Let M be a deterministic Turing machine with input alphabet Σ and let $\$, \# \notin \Sigma$. Note that the languages $L_1 = \$\Sigma^*\#$ and

$$L_2 = \{\$w\#^n \mid w \in \Sigma^*, n \geq 1, \text{ and } M \text{ accepts } w \text{ using at most } \log(n) \text{ space}\}$$

can be decided in deterministic log-space. Observe that the language $L = L_1 \cup L_2$ is complete (Lemma 2.4), hence $T(L) = T(\mu^*(L))$; no word in L_2 belongs to $T(L)$; and a word $\$w\# \in L_1$ belongs to $T(L)$ if and only if M does not accept w. Because M may accept an undecidable language, we cannot decide $T(L)$ in general. $\square$

A closure property of superposition with respect to the language class PTIME has been stated in [31]. Thus, we have:

Proposition 3. *The class* PTIME *is closed under iterated overlap assembly.*

4. Iterated overlap assembly and terminal sets

In this section we further explore the iterated overlap assembly and terminal sets. In the previous section we have seen that the terminal set of a complete context-sensitive language can be undecidable (Theorem 3.2). In this section we will show that the terminal set of a complete context-free language is always context-sensitive (Theorem 4.2). We also show that, for a context-free language L, the language $T(\mu_*(L))$ is context-sensitive (Theorem 4.3). We establish space and time complexities for deciding $T(L)$ when L is complete and given via certain automata models (Theorem 4.4 and Corollaries 4.5–4.8). Lastly, we establish a relation between Schützenberger constants and the iterated overlap assembly (Theorem 4.9).

We start with an observation about terminal sets of complete languages.

Theorem 4.1. *The terminal set of a complete language L is given by*

$$T(L) = L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$$

Proof. First, we prove $T(L) \subseteq L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$ by contradiction: suppose there were $w \in T(L) \cap \mathrm{Pref}(L) \subseteq L$. $w \in \mathrm{Pref}(L)$ implies that there is $x \in L$ such that w is a proper prefix of x. Therefore, $x \in w \overline{\odot} x \subseteq w \overline{\odot} L$ which contradicts that $w \in T(L)$ since the latter implies that $w \overline{\odot} L = w$. A similar case can be made when $w \in T(L) \cap \mathrm{Suff}(L)$.

Now, let $w \in L \setminus (\mathrm{Pref}(L) \cup \mathrm{Suff}(L))$ and suppose that $w \notin T(L)$. The latter implies that there is $x \neq w$ such that $x \in w \overline{\odot} L$ or $x \in L \overline{\odot} w$; by symmetry, we assume $x \in w \overline{\odot} L$. Note that $x \in L$ because L is complete. Also, $x \in w \overline{\odot} L$ and $x \neq w$ imply that w is a proper prefix of x. Since w is a proper prefix of $x \in L$ we obtain a contradiction. If we initially assume that $x \in L \overline{\odot} w$, we will similarly obtain a contradiction that w is a proper suffix of $x \in L$. $\square$

Note that Proposition 2 now follows as a corollary of Theorem 4.1, since regularity is preserved under the used operations. Next, we consider terminal sets of context-free languages.

Theorem 4.2. *The terminal set of a complete, context-free language L is not necessarily a context-free language, but is always a context-sensitive language.*

Proof. We first prove that $T(L)$ for a complete, context-free language L may not be context-free using a counter-example. Let

$$L = \{\#a^i b^j c^k \mid i, j, k \geq 1, k \leq i \vee k \leq j\}.$$

Note that L is clearly context-free and complete (using Lemma 2.4), but the terminal set $T(L)$ is not context-free:

$$T(L) = \{\#a^i b^j c^k \mid i, j, k \geq 1, k = \max\{i, j\}\}.$$

If L is a context-free language, then the language $M = \mathrm{Pref}(L) \cup \mathrm{Suff}(L)$ is context-free as well. Because the family of context-sensitive languages is closed under intersection [25] and complementation [32, 33], we have that $T(L) = L \cap M^c$ is context-sensitive. $\square$

Later, in Corollary 4.7, we will see that $T(L)$ is, in fact, in DSPACE($\log^2 n$) and also in DTIME($n^{2.373}$).

So far, we have seen that the iterated overlap assembly of a context-free language is context-sensitive (Proposition 1), and that the terminal set of a context-sensitive language can be undecidable (Theorem 3.2). Next we prove that, for context-free language L, the language $T(\mu_*(L))$ is context-sensitive.

Theorem 4.3. *The terminal set of $\mu_*(L)$ is context-sensitive if L is context-free.*

Proof. In order to decide $w \in T(\mu_*(L))$ we decide the two properties which are necessary and sufficient to decide the theorem statement.

1. $w \in \mu_*(L)$ and
2. $(\mathrm{pref}(w) \cap \mathrm{Suff}(L) \cap \Sigma^+) \cup (\mathrm{suff}(w) \cap \mathrm{Pref}(L) \cap \Sigma^+) = \emptyset$.

Both properties can be decided in linear space when L is context-free; if L is context-free, it is also context-sensitive and hence $\mu_*(L)$ is context-sensitive (i.e. decidable in linear space) by Proposition 1. Furthermore, it is clear that properties 1 and 2 are necessary for w to belong to $T(\mu_*(L))$. If property 2 is not true, then there is a string in $w \overline{\odot} L$ or $L \overline{\odot} w$ other than w and that falsifies that w is in $T(\mu_*(L))$.

In order to show that the conditions are sufficient, we prove the contrapositive. Consider $w \notin T(\mu_*(L))$. If $w \notin \mu_*(L)$, then condition 1 is violated. Otherwise, there exist $z \in \mu_*(L) \setminus \{w\}$ and $x \in L$ such that $z \in w \overline{\odot} x$ or $z \in x \overline{\odot} w$, by Lemma 2.3. If $z \in w \overline{\odot} x$, then $\mathrm{suff}(w) \cap \mathrm{Pref}(L) \cap \Sigma^+ \neq \emptyset$; and if $z \in x \overline{\odot} w$, then $\mathrm{pref}(w) \cap \mathrm{Suff}(L) \cap \Sigma^+ \neq \emptyset$ — hence, property 2 is violated. $\square$

Next, we investigate the terminal set $T(L)$ for various complete languages L. For convenience, we assume that all (one-way) machines have a right end marker in their read-only input tape. For nondeterministic machines, this assumption can be made without loss of generality, since such a machine can "guess" the end of the input and simulate the computation on the end marker using λ-moves at the end of the input.

Theorem 4.4. *If L is a complete language accepted by an NSCA, then $T(L)$ is in* NSPACE$(\log n)$ *and is also in* PTIME*.*

Proof. Let L be accepted by an NSCA M. We claim that Pref(L) and Suff(L) can be accepted by NSCAs. We construct an NSCA M_1 accepting Pref(L). M_1, when given input x, will accept x if there is some nonempty y such that xy is accepted by M. M_1 operates as follows: It simulates M on x faithfully. Then after processing x, M_1, on λ-moves, guesses some suffix-string y symbol-by-symbol and continues simulating the computation of M on y and accepts if M accepts. Similarly, an NSCA M_2 accepting Suff(L) can be constructed, but in this case, given input x, M_2, on λ-moves, guesses some nonempty prefix-string y and simulates M. After guessing and processing y, M_2 reads x and continues simulating M and accepts if M accepts. Clearly, from M_1 and M_2, we can also construct an NSCA M_3 accepting $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$.

It is known that every NSCA can be converted to an equivalent quasi-real time NSCA, i.e., there is a d such that during the computation, the number of consecutive λ moves on the input is bounded by d (hence the NSCA runs in linear time) [30]. We can then convert M_3 to an equivalent quasi-real time NSCA M_4. It follows that the stack-counter values during the accepting computation is linear on the length of the input. Clearly, the stack-counter can be simulated by two ordinary counters whose values would also be linear in the length of the input. Hence the stack-counter of M_4 can be stored in $\log n$ space on a read/write tape. It follows that $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$ is in NSPACE$(\log n)$. Now the complement L_3^c of L_3 is also in NSPACE$(\log n)$ [32, 33]. Since NSPACE$(\log n)$ is clearly closed under intersection, by Theorem 4.1, $T(L) = L \setminus (\text{Pref}(L) \cup \text{Suff}(L)) = L \cap L_3^c$ is also in NSPACE$(\log n)$. That $T(L)$ is in PTIME follows from the fact that NSPACE$(\log n) \subseteq$ PTIME. □

We can use a similar construction as in the proof of Theorem 4.4 to obtain the following results.

Corollary 4.5. *If L is a complete language accepted by an NCA, then $T(L)$ is in* NSPACE$(\log n)$ *and* DTIME(n^2).

Proof. $T(L)$ in NSPACE$(\log n)$ follows from Theorem 4.4 since NCA is a special case of NSCA. The time complexity follows from the proof of Theorem 4.4 and the fact that every language accepted by an NCA is in DTIME(n^2) [34] and that this class is closed under complementation and intersection. □

Corollary 4.6. *If L is a complete language accepted by an NFCM, then $T(L)$ is in* NSPACE$(\log n)$ *and also in* PTIME.

Proof. If L is accepted by an NFCM M, then, as in the proof of Theorem 4.4, we can construct an NFCM M_3 accepting $L_3 = \text{Pref}(L) \cup \text{Suff}(L)$. It is known that for any NFCM, there is a fixed constant d such that any string of length n accepted by the NFCM can be accepted in dn steps, i.e., the values of the counters are at most dn [35]. It follows that any NFCM language is in NSPACE$(\log n)$. Then, as in the proof of Theorem 4.4, $T(L) = L \cap L_3^c$ is in NSPACE$(\log n)$ and, hence, also in PTIME. □

The next corollary strengthens Theorem 4.2.

Corollary 4.7. *If L is a complete language accepted by an NPDA (i.e., L is a complete context-free language), then $T(L)$ is in* DSPACE$(\log^2 n)$ *and* DTIME$(n^{2.373})$ *(= complexity of matrix multiplication).*

Proof. This follows by similar constructions as in Theorem 4.4 using the fact that every language accepted by an NPDA is in DSPACE$(\log^2 n)$ and also in DTIME$(n^{2.373})$ (= complexity of matrix multiplication [36]), and the fact that these classes are closed under complementation and intersection. □

Similarly, since the family of linear context-free languages (i.e., languages accepted by 1-reversal NPDAs)is in DTIME(n^2) [37], we have:

Corollary 4.8. *If L is a complete linear context-free language, then $T(L)$ is in* DTIME(n^2).

Lastly, we consider the relation between Schützenberger constants [38] and the iterated overlap assembly. A word $w \in \Sigma^+$ is a *(Schützenberger) constant* for L if $w \in \inf(L)$ and for all words $u_1, u_2, v_1, v_2 \in \Sigma^*$, we have that

$$u_1 w v_1 \in L \text{ and } u_2 w v_2 \in L \implies u_1 w v_2 \in L.$$

The existence of constants in a language seems to have a close connection to languages that are generated by some biologically inspired systems; for example, every splicing language has a constant [39].

Theorem 4.9. *Every word $w \in \Sigma^+$ in $\mu_*(L) \setminus \mathrm{Inf}(L)$ is a constant for $\mu_*(L)$. If, in addition, w satisfies $w \in \inf(T(\mu_*(L)))$, then w is a constant for $T(\mu_*(L))$ as well.*

Proof. Let $w \in \mu_*(L) \setminus \mathrm{Inf}(L)$ and let $u_1, v_1, u_2, v_2 \in \Sigma^*$ such that $u_1wv_1 \in \mu_*(L)$ and $u_2wv_2 \in \mu_*(L)$. In order to show that w is a constant, we show that $u_1w \in \mu_*(L)$ and $wv_2 \in \mu_*(L)$; this implies that $u_1wv_2 \in u_1w \overline{\odot} wv_2 \subseteq \mu_*(L)$.

Let x_1 be the longest prefix of u_1w that belongs to $\mu_*(L)$. If u_1 is a proper prefix of x_1, then $u_1w \in x_1 \overline{\odot} w \subseteq \mu_*(L)$. Otherwise, x_1 would be prefix of u_1 which is not possible: Suppose x_1 is a prefix of u_1 and let x_2 be such that $x_1x_2 = u_1wv_1$. Lemma 2.2 implies that there are $y_1, y_2 \in \Sigma^+$ with $y_1y_2 \in L$ such that y_1 is a suffix of x_1 and y_2 is a prefix of x_2. Since w is no proper infix of a word in L, it is no infix of y_2 either. We obtain that $x_1y_2 \in x_1 \overline{\odot} y_1y_2 \subseteq \mu_*(L)$ is a prefix of u_1w; this contradicts the choice of x_1 which is supposed to be the longest prefix with that property. By a symmetric argument, we can show that $wv_2 \in \mu_*(L)$. We conclude that w is a constant for $\mu_*(L)$.

Now, consider the case when $u_1wv_1 \in T(\mu_*(L))$ and $u_2wv_2 \in T(\mu_*(L))$ and, hence, $w \in \inf(T(\mu_*(L)))$. As we showed above, $u_1wv_2 \in \mu_*(L)$. In order to obtain a contradiction, suppose that $u_1wv_2 \notin T(\mu_*(L))$. By Lemma 2.3 there exist $z \in \mu_*(L) \setminus \{u_1wv_2\}$ and $x \in L$ such that $z \in u_1wv_2 \overline{\odot} x$ or $z \in x \overline{\odot} u_1wv_2$. Due to symmetry, we only consider the case when $z \in u_1wv_2 \overline{\odot} x$. Let y be the nonempty suffix of x such that $z = u_1wv_2y$. Because x cannot have w as proper infix, we have $u_2wv_2y \in u_2wv_2 \overline{\odot} x$ which contradicts the premise $u_2wv_2 \in T(\mu_*(L))$. □

5. Decision problems

In this section we consider three decision problems: whether a language is complete (Subsection 5.1), whether a string is terminal with respect to a language (Subsection 5.2), and whether the overlap assembly of two given languages equals a given third one (Subsection 5.3).

5.1. Deciding the completeness of a language

The problem of deciding if a given language is complete was studied in [1] for language classes in Chomsky hierarchy. In this subsection we narrow the gap between the language classes whose completeness is decidable and those for which it is undecidable. Recall first a result from [1]:

Proposition 4 ([1])**.**

1. *It is decidable if any given regular language is complete.*
2. *It is undecidable if any given context-free(resp., context-sensitive, recursively enumerable) language is complete.*

The following shows that Proposition 4, part 1 holds for DFCMs (i.e., DFAs augmented with reversal bounded counters).

Theorem 5.1. *It is decidable, given a DFCM M, if $L(M)$ is complete.*

Proof. Given a DFCM M accepting L, we construct an NFCM M' accepting $L' = L \overline{\odot} L$ as follows:

M', when given input z, guesses a partition $z = uvw$ for some u, v, w with $v \neq \lambda$, and checks that uv is in L by running M on uv, and vw is in L by running another copy of M on vw, and accepts z if and only if M accepts uv and vw. Note that M' uses two sets of counters of M to simulate the two copies of M. Clearly, $L(M') = L \overline{\odot} L$. It follows that $L \overline{\odot} L = L$ if and only if $L(M') \subseteq L$, and if and only if $L(M') \cap L^c = \emptyset$. Since the family of DFCM languages is effectively closed under complementation, we can construct from M a DFCM accepting L^c [27]. The result follows, since we can construct, given two NFCMs, an NFCM accepting their intersection language, and emptiness of NFCMs is decidable [27]. □

In contrast to Theorem 5.1, for the case of NFCM we have the following result which strengthens Proposition 4, part 2:

Theorem 5.2. *It is undecidable, given a 1-reversal NCA (i.e., an NFA augmented with one counter which makes only 1 reversal) M, whether $L(M)$ is complete.*

Proof. We reduce the problem to the undecidability of the halting problem for deterministic Turing machines (DTMs) on an initially blank tape.

Let Z be single-tape DTM. Without loss of generality, we assume that if Z halts on an initially blank tape, it makes at least two moves. Assume that Z does not write blanks. A configuration (ID) of Z can be represented by a string xqy, where xy is the non-zero content of the tape and q is the state of the TM. The initial configuration, $ID_1 = q_0$ (the initial state). We may assume that the TM halts in a unique halting state f. Thus the halting ID is of the form xfy for some x, y.

We construct a 1-reversal NCA M which accepts the language:

> $L(M) = \{w \mid w \neq ID_1 \# ID_2 \cdots \# ID_k$, where $k \geq 3, ID_1$ is the initial configuration of Z on an initially blank tape, ID_k is a halting configuration of Z, and ID_{i+1} is the valid successor of $ID_i, 1 \leq i \leq k-1\}$.

The construction of M is straightforward (see, e.g., [35]). For completeness, we briefly sketch it here. Given input w, M accepts w if
(1) w is not of the format above, or
(2) ID_1 is not the initial configuration, or
(3) ID_k is not a halting configuration, or
(3) ID_{i+1} is not the valid successor of ID_i for some i.

Clearly, M's finite-state control can check (1), (2), (3). To check (4), M nontermiministically moves its input head to the beginning of some nondeterministically chosen ID_i andrecords a nondeterministically chosen position d within ID_i by incrementing the counter to value d. Then M moves to the beginning of ID_{i+1} and decrements the counter to zero to find the corresponding position in ID_{i+1}. By doing this, M can detect if there is a discrepancy in the symbols in the three positions $d-1, d, d+1$ of ID_{i+1} with respect to the symbols in the corresponding positions in ID_i.

Let Σ be the alphabet over which $L(M)$ is defined. Clearly, $L(M) = \Sigma^*$ (which is complete) if Z does not halt on blank tape. However, if Z halts on blank tape, $L(M) = \Sigma^* \setminus \{x\}$ for exactly one string x of the form $ID_1 \# ID_2 \# \cdots \# ID_k$, $k \geq 3$, and it is not complete because: $ID_1 \# ID_2$ is in $L(M)$ (since it is not x) and $ID_2 \# \cdots \# ID_k$ is also in $L(M)$ (since it is not x). Hence, $x = ID_1 \# ID_2 \# \cdots \# ID_k$ is in $L(M) \overline{\odot} L(M)$, but it is not in $L(M)$. It follows that $L(M)$ is complete if and only if Z does not halt on blank tape, which is undecidable. □

5.2. Deciding the terminality of strings

We now investigate the problem of deciding whether a given string is terminal with respect to a language. The following result gives sufficient conditions for the decidability of whether a string w is terminal with respect to a language.

Theorem 5.3. *Let $\mathcal{M}$ be a class of machines, and let $\mathcal{L}$ be the corresponding class of accepted languages, satisfying:*

1. *if L is in $\mathcal{L}$, then for any string w in L, $w \overline{\odot} L$ and $L \overline{\odot} w$ are also in $\mathcal{L}$;*
2. *$\mathcal{L}$ is closed under intersection with regular sets;*
3. *$\mathcal{L}$ has a decidable emptiness problem;*

and items 1 and 2 are effective. Then it is decidable, given a machine M in $\mathcal{M}$ and a string w in $L(M)$, if w is terminal with respect to $L(M)$.

Proof. Let L be a language accepted by a machine $M \in \mathcal{M}$ and w be a string in L. Then, by item 1, we can construct machines in $\mathcal{M}$ accepting $L_1 = w \overline{\odot} L$ and $L_2 = L \overline{\odot} w$. To check that $L_1 = \{w\}$, we do the following: since $\mathcal{L}$ is closed under intersection with regular sets (item 2) and has a decidable emptiness problem (item 3), we check that $L_1 \cap \{w\}^c = \emptyset$ (note that $w \in L_1$ is always true). Similarly, we can check that $L_2 = \{w\}$. $\square$

Almost all classes of one-way nondeterministic machines satisfy condition 1 in Theorem 5.3. Indeed, given M accepting L and $w \in L$, we construct a machine M' accepting $w \overline{\odot} L$ which, on a given input z, guesses a decomposition of z into uvx and checks that that $uv = w$ and vx is accepted by M. Similarly, we can construct a machine M'' to accept $L \overline{\odot} w$.

As examples, the classes of languages accepted by NPCMs and NSAs satisfy condition 1 of Theorem 5.3, while condition 2 is also clearly satisfied. Since emptiness for NPCMs and NSAs is decidable [27, 29], we have:

Corollary 5.4. *It is decidable, given an NPCM (resp., NSA) M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

Next we show that condition 3 in Theorem 5.3 is a necessary condition. We say that a class of languages $\mathcal{L}$ is closed under *distinct-symbol concatenation* if, given $L \in \mathcal{L}$ and a symbol \$, not in the alphabet of L, $\$L$ and $L\$$ are in $\mathcal{L}$.

Theorem 5.5. *Let $\mathcal{M}$ be a class of machines, and $\mathcal{L}$ be the corresponding class of accepted languages. Assume that $\mathcal{L}$ is effectively closed under distinct-symbol concatenation and union with a singleton language. If $\mathcal{L}$ has an undecidable emptiness problem, then it is undecidable, given a language L in $\mathcal{L}$ and a string w in L, whether w is terminal with respect to L.*

Proof. Let M_1 be a machine in $\mathcal{M}$ accepting a language $L_1 \subseteq \Sigma^*$. Let $\%, \#, \$$ be new symbols not in Σ. Consider the string $w = \%\#$. Construct a machine M in $\mathcal{M}$ accepting the language $L = \{\%\#\} \cup \{\%\#x\$ \mid x \in L_1\}$. Clearly, $\%\# \mathbin{\overline{\odot}} L = L \mathbin{\overline{\odot}} \%\# = \{\%\#\}$ if and only if $L_1 = \emptyset$. We cannot decide if w is terminal, since emptiness for $\mathcal{L}$ is undecidable. □

An example of a class $\mathcal{L}$ such as the one in Theorem 5.5 is the class of languages accepted by real-time DFAs augmented with two unrestricted counters (real-time deterministic 2-counter machines). Real-time here means that the machines have no λ-moves. It is known that it is undecidable, given a deterministic machine Z which has *no input tape* but with two counters that are initially zero, whether it will halt [26]. We construct from Z, a real-time deterministic 2-counter machine M which (unlike Z) has an input tape. M, when given a unary input string a^n, simulates Z's counters faithfully while reading a symbol a (of the input) on each move of Z, and accepts if and only if Z halts after reading exactly n a's. Clearly if Z halts after n steps, M will accept a^n. If Z does not halt, M will not accept a^n for any n. Hence, $L(M) = \emptyset$ if and only if Z does not halt, and M operates in real-time. It follows that the emptiness problem for real-time deterministic 2-counter machines is undecidable. Now it is obvious that if L is accepted by a real-time deterministic 2-counter machine M and $\$$ is a new symbol, then $\$L$ and $L\$$ can also be accepted by real-time deterministic 2-counter machines. If $\{x\}$ is a singleton language, then $L \cup \{x\}$ can be accepted by a real-time deterministic 2-counter machine M' which simulates M and also incorporates in the finite-state control the string x; so M' can detect and accept if the input is x. Hence, the assumptions in Theorem 5.5 are satisfied, and we have:

Corollary 5.6. *It is undecidable, given a real-time deterministic 2-counter machine M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

As above, we can also construct a (one-way) real-time deterministic $\log n$ space-bounded DTM to simulate Z. Hence, the emptiness problem for these machines is also undecidable. Thus, we have:

Corollary 5.7. *It is undecidable, given a real-time deterministic* $\log n$ *space-bounded DTM M and a string w in $L(M)$, whether w is terminal with respect to $L(M)$.*

5.3. Deciding the given decomposition of a language

Finally, we consider the problem of deciding, given languages L, L_1, L_2, whether $L = L_1 \overline{\odot} L_2$.

Theorem 5.8.

1. *It is undecidable, given a language L accepted by a 1-reversal NCA and regular languages L_1 and L_2, whether $L = L_1 \overline{\odot} L_2$.*
2. *It is undecidable, given a regular language L and languages L_1 and L_2 accepted by 1-reversal DPDAs (resp., DCAs), whether $L = L_1 \overline{\odot} L_2$.*

Proof. For part 1, let $L \subseteq \Sigma^+$ be accepted by a 1-reversal NCA and let $L_1 = L_2 = \Sigma^+$. Hence, $L_1 \overline{\odot} L_2 = \Sigma^+$. The result follows, since it is undecidable whether the language accepted by a 1-reversal NCA is equal to Σ^+ (as seen in the proof of Theorem 5.2).

For part 2, let $L_1', L_2' \subseteq \Sigma^+$ be accepted by 1-reversal DPDAs (resp., DCAs). Let $\#, \$$ be new symbols not in Σ. Let $L = \{\$\$\}$, $L_1 = \#L_1'\$ \cup \{\$\$\}$, and $L_2 = \#L_2'\$ \cup \{\$\$\}$ Clearly, L_1 and L_2 can also be accepted by 1-reversal DPDAs (resp., DCAs). Then $L = L_1 \overline{\odot} L_2$ if and only if $L_1' \cap L_2' = \emptyset$. The result now follows since it is undecidable if the intersection of two languages accepted by 1-reversal DPDAs (resp., DCAs) is empty [25, 27]. □

In contrast to Theorem 5.8, part 2, when L is accepted by a deterministic machine we have the following result.

Theorem 5.9. *It is decidable, given a language L accepted by a DFCM (resp., DPCM) and regular languages L_1 and L_2, whether $L = L_1 \overline{\odot} L_2$.*

Proof. Clearly, $L_3 = L_1 \overline{\odot} L_2$ is regular. Now $L = L_1 \overline{\odot} L_2$ if and only if $L \cap L_3^c = \emptyset$, and $L^c \cap L_3 = \emptyset$. The result follows since the class of DFCM (resp., DPCM) languages is closed under intersection with regular sets and complementation, and has a decidable emptiness problem [27, 40]. □

Finally, in contrast to Theorem 5.8, when the problem concerns "containment", we have:

Theorem 5.10.

1. *It is decidable, given languages L_1 and L_2 accepted by NFCMs and a language L accepted by a DPCM, whether $L_1 \overline{\odot} L_2 \subseteq L$.*
2. *It is decidable, given a language L accepted by an NPCM and regular languages L_1 and L_2, whether $L \subseteq L_1 \overline{\odot} L_2$.*

Proof.

1. Clearly, $L_1 \overline{\odot} L_2 \subseteq L$ if and only if $(L_1 \overline{\odot} L_2) \cap L^c = \emptyset$ (where L^c is the complement of L). If L_1 and L_2 are accepted by NFCMs, we can effectively construct an NFCM accepting $L_1 \overline{\odot} L_2$ [2]. If L is accepted by a DPCM, we can effectively construct a DPCM accepting L^c. Since the languages accepted by NPCMs is effectively closed under intersection with languages accepted by NFCMs [27], and the emptiness problem for languages accepted by NPCMs is decidable, the result follows.

2. If L_1 and L_2 are regular languages accepted by DFAs, we can effectively construct a DFA accepting $L_1 \overline{\odot} L_2$ [2]. The result follows since the complement of a regular language is regular, NPCM languages are effectively closed under intersection with regular sets [27], and the emptiness problem for NPCMs is decidable [27].

□

6. Concluding remarks

This paper continues the exploration, started in [1] and [2], of the properties of the overlap assembly operation. In particular, it strengthens the results given in [1] regarding the closure of language classes under iterated overlap assembly and the decidability of the completeness of a language. It also enhances the results regarding closure properties of terminating sets of languages (which are almost equivalent to *maximal (adult) languages* in [22, 31]). Finally, it investigates the problem of deciding whether a given string is terminal with respect to a language, and the problem of deciding if a given language can be generated by an overlap assembly operation of two given others. Further directions of research include investigations of decision problems such as those studied in Section 5.3 for various other language classes, and finding an efficient algorithm that, given a language L, outputs a pair of languages (if they exist) whose overlap assembly equals L.

Acknowledgements

We thank the reviewers for their comments which improved the presentation of our results.

References

[1] E. Csuhaj-Varjú, I. Petre, G. Vaszil, Self-assembly of strings and languages, Theoretical Computer Science 374 (1-3) (2007) 74–81.

[2] S. K. Enaganti, O. H. Ibarra, L. Kari, S. Kopecki, On the overlap assembly of strings and languages, Natural Computing (2016) 1–11.

[3] D. Cheptea, C. Martín-Vide, V. Mitrana, A new operation on words suggested by DNA biochemistry: hairpin completion, in: Proc. Transgressive Computing, TC, 2006, pp. 216–228.

[4] F. Manea, V. Mitrana, Hairpin completion versus hairpin reduction, in: S. B. Cooper, B. Löwe, A. Sorbi (Eds.), Proc. Computability in Europe, CiE, Vol. 4497 of LNCS, 2007, pp. 532–541.

[5] F. Manea, C. Martín-Vide, V. Mitrana, On some algorithmic problems regarding the hairpin completion, Discrete Applied Mathematics 157 (2009) 2143–2152.

[6] S. Kopecki, On iterated hairpin completion, Theoretical Computer Science 412 (29) (2011) 3629–3638.

[7] C. Martín-Vide, G. Păun, J. Pazos, A. Rodríguez-Patón, Tissue P systems, Theoretical Computer Science 296 (2) (2003) 295–326.

[8] S. K. Enaganti, L. Kari, S. Kopecki, A formal language model of DNA polymerase activity, Fundamenta Informaticae 138 (2015) 179–192.

[9] W. P. Stemmer, DNA shuffling by random fragmentation and reassembly: in vitro recombination for molecular evolution, Proceedings of the National Academy of Sciences 91 (22) (1994) 10747–10751.

[10] P. D. Kaplan, Q. Ouyang, D. S. Thaler, A. Libchaber, Parallel overlap assembly for the construction of computational DNA libraries, Journal of Theoretical Biology 188 (3) (1997) 333–341.

[11] Q. Ouyang, P. D. Kaplan, S. Liu, A. Libchaber, DNA solution of the maximal clique problem, Science 278 (5337) (1997) 446–449.

[12] A. R. Cukras, D. Faulhammer, R. J. Lipton, L. F. Landweber, Chess games: a model for RNA based computation, Biosystems 52 (1-3) (1999) 35–45.

[13] D. Faulhammer, A. R. Cukras, R. J. Lipton, L. F. Landweber, Molecular computation: RNA solutions to chess problems, Proceedings of the National Academy of Sciences 97 (4) (2000) 1385–1389.

[14] G. Franco, C. Giagulli, C. Laudanna, V. Manca, DNA extraction by XPCR, in: C. Ferretti, G. Mauri, C. Zandron (Eds.), Proc. DNA Computing, (DNA 11), Vol. 3384 of LNCS, 2005, pp. 104–112.

[15] G. Franco, V. Manca, C. Giagulli, C. Laudanna, DNA recombination by XPCR, in: A. Carbone, N. A. Pierce (Eds.), Proc. DNA Computing, (DNA 12), Vol. 3892 of LNCS, 2006, pp. 55–66.

[16] V. Manca, G. Franco, Computing by polymerase chain reaction, Mathematical Biosciences 211 (2) (2008) 282–298.

[17] G. Franco, A polymerase based algorithm for SAT, in: M. Coppo, E. Lodi, G. Pinna (Eds.), Theoretical Computer Science, Vol. 3701 of LNCS, Springer Berlin Heidelberg, 2005, pp. 237–250.

[18] G. Franco, V. Manca, Algorithmic applications of XPCR, Natural Computing 10 (2) (2011) 805–819.

[19] E. Winfree, X. Yang, N. Seeman, Universal computation via self-assembly of DNA: Some theory and experiements, in: L. Landweber, E. Baum (Eds.), Proc. DNA Computing, (DNA 2), Vol. 44 of DIMACS, 1998, pp. 191–213.

[20] E. Winfree, Algorithmic self-assembly of DNA, Ph.D. thesis, California Institute of Technology (1998).

[21] E. Winfree, T. Eng, G. Rozenberg, String tile models for DNA computing by self-assembly, in: A. Condon, G. Rozenberg (Eds.), Proc. DNA Computing, (DNA 6), Vol. 2054 of LNCS, 2001, pp. 63–88.

[22] P. Bottoni, A. Labella, V. Manca, V. Mitrana, Superposition based on Watson-Crick-like complementarity, Theory of Computing Systems 39 (4) (2006) 503–524.

[23] L. Kari, R. Kitto, G. Thierrin, Codes, involutions, and DNA encodings, in: W. Brauer, H. Ehrig, J. Karhumäki, A. Salomaa (Eds.), Formal and Natural Computing, Vol. 2300 of LNCS, 2002, pp. 376–393.

[24] S. Hussini, L. Kari, S. Konstantinidis, Coding properties of DNA languages, in: N. Jonoska, N. C. Seeman (Eds.), Proc. DNA Computing, (DNA 7), Vol. 2340 of LNCS, 2002, pp. 57–69.

[25] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Inc., 1978.

[26] M. L. Minsky, Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines, The Annals of Mathematics 74 (3) (1961) 437–455.

[27] O. H. Ibarra, Reversal-bounded multicounter machines and their decision problems, J. ACM 25 (1) (1978) 116–133.

[28] O. H. Ibarra, Automata with reversal-bounded counters: a survey, in: Proc. Descriptional Complexity of Formal Systems, DCFS, 2014, pp. 5–22.

[29] S. Ginsburg, S. A. Greibach, M. A. Harrison, One-way stack automata, J. ACM 14 (2) (1967) 389–418.

[30] S. Ginsburg, G. F. Rose, The equivalence of stack-counter acceptors and quasi-realtime stack-counter acceptors, Journal of Computer and System Sciences 8 (2) (1974) 243–269.

[31] F. Manea, V. Mitrana, J. Sempere, Some remarks on superposition based on Watson-Crick-like complementarity, in: V. Diekert, D. Nowotka (Eds.), Developments in Language Theory, Vol. 5583 of LNCS, 2009, pp. 372–383.

[32] N. Immerman, Nondeterministic space is closed under complementation, SIAM J. Comput. 17 (5) (1988) 935–938.

[33] R. Szelepcsényi, The method of forced enumeration for nondeterministic automata, Acta Informatica 26 (3) (1988) 279–284.

[34] S. A. Greibach, A note on the recognition of one counter languages, Informatique Théorique et Applications 9 (2) (1975) 5–12.

[35] B. S. Baker, R. V. Book, Reversal-bounded multipushdown machines, Journal of Computer and System Sciences 8 (3) (1974) 315–332.

[36] V. V. Williams, Multiplying matrices faster than Coppersmith-Winograd, in: Proc. ACM Symposium on Theory of Computing, STOC, 2012, pp. 887–898.

[37] T. Kasami, A note on computing time for recognition of languages generated by linear grammars, Information and Control 10 (2) (1967) 209–214.

[38] M.-P. Schützenberger, Sur certaines opérations de fermeture dans les langages rationnels, in: Symposia Mathematica, Vol. 15, 1975, pp. 245–253.

[39] P. Bonizzoni, N. Jonoska, Existence of constants in regular splicing languages, Information and Computation 242 (2015) 340–353.

[40] O. H. Ibarra, H.-C. Yen, On the containment and equivalence problems for two-way transducers, Theoretical Computer Science 429 (2012) 155–163.